Hybrid Genetic Algorithms for Bin-packing and Related Problems *

Colin Reeves
School of Mathematical and Information Sciences
Coventry University
UK
Email: CRReeves@coventry.ac.uk

Abstract

The genetic algorithm (GA) paradigm has attracted considerable attention as a promising heuristic approach for solving optimization problems. Much of the development has related to problems of optimizing functions of continuous variables, but recently there have been several applications to problems of a combinatorial nature.

What is often found is that GAs have fairly poor performance for combinatorial problems if implemented in a naive way, and most reported work has involved somewhat *ad hoc* adjustments to the basic method.

In this paper, we will describe a general approach which promises good performance for a fairly extensive class of problems by hybridizing the GA with existing simple heuristics. The procedure will be illustrated mainly in relation to the problem of *bin-packing*, but it could be extended to other problems such as *graph-partitioning, parallel-machine scheduling* and *generalized assignment*. The method is further extended by using *problem size reduction* hybrids. Some results of numerical experiments will be presented which attempt to identify those circumstances in which these heuristics will perform well relative to exact methods.

Finally, we discuss some general issues involving hybridization: in particular, we raise the possibility of blending GAs with orthodox mathematical programming procedures.

1 Introduction

Since the introduction and exploration of the concept of computational complexity [1, 2], it has become widely recognized that solving NP-hard combinatorial optimization problems (COPs) to optimality in general is impracticable, although in certain cases (e.g. knapsack problems) optimal solutions to quite large problem instances can be found fairly routinely. However, recent years have seen significant advances in developing heuristic techniques for finding high-quality solutions to such problems. Some of these approaches are described in [3], including such methods as *simulated annealing*, *tabu search* and the subject of this paper—*genetic algorithms* (GAs).

While much of the early work on applying GAs to optimization problems related to the solution of continuous problems, increasing attention is being given to applications to the types of discrete optimization problem that are of interest in Operational Research (OR). Perhaps inevitably, the first case to be extensively investigated was the travelling salesman problem (TSP) [4, 5, 6, 7], with reasonable but not spectacular results. Other problems investigated include machine sequencing [8, 9], set-covering [10], Steiner trees [11] and bin-packing [12, 13]. In many, although not all, cases the simple GA was found to be a relatively ineffective heuristic,

*Published in *Annals of OR*, **63**, 371-396.

It is clear that their successful application to COPs will nearly always need some problem-specific information. Any such procedure can be characterized as a 'hybrid' GA, although in this paper we shall use the term to mean a procedure which integrates some other heuristic into the framework of a genetic search.

In what follows, we first describe, very briefly, the chief features of a simple genetic algorithm, and then discuss some of the reasons why such an approach is often unsatisfactory for a COP. In this regard, we will focus particularly on the problem of bin-packing, where we shall develop a line of attack which has potential in a number of other problems. We also report the results of some numerical experiments which were carried out to investigate the effectiveness of the hybrid approach adopted.

2 The simple genetic algorithm

From an OR perspective, the idea of a genetic algorithm can be understood as the intelligent exploitation of information gained in a random search. A comprehensive introduction to GAs in the OR context can be found in [3], while Goldberg [14] gives an excellent general presentation.

Here, we simply sketch in the basic ideas. The name *genetic algorithm* originates from the analogy between the representation of a complex structure by means of a vector of components, and the idea, familiar to biologists, of the genetic structure of a chromosome. In selective breeding of plants or animals, for example, offspring are sought which have certain desirable characteristics—characteristics which are determined at the genetic level by the way the parents' chromosomes combine. In a similar way, in seeking better solutions to COPs, we often intuitively re-combine pieces of existing solutions.

In many applications, the component vector is simply a string of 0s and 1s, and although it is not a necessary characteristic of a GA, much of the theoretical development is easier to understand from this perspective. Many *genetic operators* have been identified, the most commonly used ones being *crossover* (an exchange of sections of the parents' chromosomes), and *mutation* (a random modification of the chromosome).

In the context of most obvious relevance to OR, that of finding the optimal solution to a large COP, a Genetic Algorithm works by maintaining a population of M solutions—potential *parents*— whose *fitness values*[1] have been calculated. In Holland's original GA [15], two parents are chosen, at least one of which is selected on the basis of its fitness (the better the fitness value, the greater the chance of it being chosen). They are then *mated* by choosing a crossover point X at random, the offspring consisting of the 'left-hand' section from one parent followed by the 'right-hand' section of the other.

For example, suppose we have parents P1 and P2 as follows, with crossover point X; then the offspring will be the pair O1 and O2:

```
P1   1 0 1 0 0 1 0      O1   1 0 1 1 0 0 1
           X
P2   0 1 1 1 0 0 1      O2   0 1 1 0 0 1 0
```

One of the existing population is chosen at random, and replaced by one of the offspring. This reproductive plan is repeated as many times as is desired. In other versions of this procedure, parents may be chosen in strict proportion to their fitness values (rather than probabilistically), or the whole population may be changed after every set of M trials, rather than incrementally. Different types of crossover operator may be defined, with $m(>1)$ crossover points, where m is either be fixed in advance, or chosen stochastically ('uniform' crossover). Other operators are

[1] Fitness is generally measured by some monotonic function of the associated objective function values.

random changes (at a low rate) into the population.

The concept Holland developed to help understand the process of a GA was that of a *schema*, which is a subset of the universe of all solutions, defined by the possession of a common characteristic. For instance, the two vectors

```
1 0 1 0 0 0 1
0 1 1 1 0 1 0
```

are both examples of the schema

```
* * 1 * 0 * * .
```

Holland was able to show, firstly, that the expected representation of a given schema in the next generation rises or falls in proportion to its fitness in the current generation (a result commonly known as the Schema Theorem), and secondly, that under certain conditions a population of M chromosomes will contain information on $\mathcal{O}(M^3)$ schemata. It is these two facts that are commonly used to explain why it is that GAs provide an effective and efficient means of finding near-optimal solutions for a wide variety of problems. There is also a third assumption implicit in the implementation of a GA: that the re-combination of small pieces (fit 'low-order' schemata) into bigger pieces is indeed a sensible method of finding the optimal solution to such problems. Goldberg [14] calls this the 'building-block' hypothesis, and it is this that often causes difficulties for combinatorial problems.

3 GAs for combinatorial optimization

The application of genetic algorithms to combinatorial problems often leads to some specific difficulties which arise much less often in the cases of numerical optimization reported in the GA literature. A more detailed discussion can be found in [16], but here we describe some of these difficulties as they arise in the context of one-dimensional bin-packing.

3.1 Binary coded COPs

Many combinatorial problems can be formulated in terms of a $0/1$ integer-programming problem. In such cases, the application of a GA would appear to be straightforward, as the encoding of a 'solution' to the problem as a binary string is obvious. However, the existence of constraints in such cases causes difficulties, as many strings will in fact encode infeasible solutions to the problem, and how to assign a 'fitness' value to such strings is not at all clear. Richardson *et al.* [10] give some guidance on the use of penalties to deal with constraints, but a completely satisfactory solution to this problem has not yet been achieved, and a full discussion is outside the scope of this paper. The example of bin-packing will suffice to illustrate some of the difficulties.

In the case of (one-dimensional) bin-packing, we have the problem of assigning n objects to bins of identical size in such a way as to minimize the number of bins used. Thus, we could define a chromosome of length nq, where q is an upper bound on the optimal number of bins, and

$$\begin{aligned} x_{ij} &= 1 \text{ if object } i \text{ is in bin } j \\ &= 0 \text{ otherwise.} \end{aligned}$$

This would enable traditional crossover to be used, but at the expense of a rather lengthy chromosome. Many of the solutions would almost inevitably be infeasible, and penalty functions would be needed.

$$
\begin{aligned}
x_{ij} &= 1 \text{ if object } i \text{ is in the same bin as object } j \\
&= 0 \text{ otherwise.}
\end{aligned}
$$

In other words, the problem is re-interpreted as finding the minimal number of equivalence classes defined by the bin-packing size constraint. There are clear difficulties with this approach too. Firstly, the number of bins implied by each 'solution' is not obvious, and would need a subsidiary calculation. Secondly, crossover may destroy the 'transitivity' property of a solution (i.e. $x_{ij} = x_{jk} = 1 \Rightarrow x_{ik} = 1$); this would have to be restored after each recombination. Thirdly, infeasibilities are likely to arise, and finally, the chromosome is likely to be even longer than in the previous case.

These difficulties would seem to rule out the use of a traditional GA; at any rate no promising results using such an approach have been reported.

3.2 q-ary coding

Early work on GAs had emphasized binary coding, virtually to the exclusion of any other representation. However, recently Radcliffe [23] and others [24, 25] have made a strong case for the appropriate use of non-binary codings.

Certainly, in many problems binary coding would not appear to be a sensible procedure, and a q-ary coding—i.e. using an alphabet of $q(>2)$ characters—is more appropriate. However, in COPs, the same type of difficulties are likely to arise as in the binary case. For example, in bin-packing, a 'natural' coding would appear to be to label the bins from 1 to q, and to assign each object a bin number. However, even in fairly trivial cases this may lead to quite severe difficulties. For instance, consider the following situation, where we have 7 objects:

```
P1   1 2 1 3 2 2 1      O1   1 2 1 3 1 1 4
          X
P2   4 1 4 3 1 1 4      O2   4 1 4 3 2 2 1
```

Here, P1 and P2 actually represent the same solution (labelling of the bins is arbitrary, given identical bins), each using 3 bins. Yet traditional crossover is quite oblivious to this fact, and actually creates two different solutions with 4 bins. Thus, recombination of 'good' solutions can easily lead to inferior ones, while it is equally obvious that recombination of feasible solutions can often lead to infeasible ones.

Falkenauer and Delchambre [13] have discussed the use of a q-ary coding for bin-packing, and pointed out some other defects of such a representation. A potentially 'good' solution should ideally have those objects which are assigned to the same bin in close proximity in the solution string, in order to prevent disruption by crossover. This could be achieved by using an auxiliary operator called 'inversion', but ultimately to no avail, since as 'better' strings are found, the number of objects assigned to the same bin should increase (the schemata get longer), only for such solutions to be disrupted by crossover. Mutation is no help as it is too destructive of good solutions.

The solution adopted in [13] was to focus not on the objects, but on the bins. The chromosome is in two parts: the first, of length n, uses a q-ary encoding of the bin identifiers as discussed above, but the second encodes the actual bin identifiers used. Thus the complete chromosome may vary in length, depending on how many bins are actually needed. Crossover takes place only on the second part of the chromosome; this normally leads to both under- and over-specification with respect to the bin contents, which is dealt with principally by using a repair mechanism based on the *first-fit descending* heuristic (FFD—about which see further below). In addition,

operator.

Finally, Falkenauer and Delchambre discuss the nature of the fitness function to be used. It is clearly of little help simply to use the number of bins as a fitness measure, as this fails to discriminate between similar solutions, of which there are likely to be many in any population. They propose using the following measure for a particular solution S

$$f(S) = \frac{\sum_{i=1}^{N}(C_i/C_{max})^2}{N}$$

where N is the number of bins actually needed by S, C_i is the actual amount allocated to bin i, and C_{max} is the maximum bin capacity. This formulation reflects the fact that the fuller a bin is, the more room there is for the remaining objects in other bins, so increasing the chances of using fewer bins altogether.

They reported very satisfactory results for their approach. The test problem instances used randomly generated object sizes and had known optimal solutions; they also had different grades of difficulty. The GA found optimal solutions almost always for the easier problems, and a fair proportion of the time even for very hard ones.

3.3 Permutation-coded COPs

Not all COPs have a natural 0/1 interpretation; there is an important class of problem for which a permutation is the 'obvious' representation. The most well-known member of this class is of course the TSP; it also includes problems such as quadratic assignment, and a multitude of machine sequencing problems. Such problems present an immediate difficulty for a GA approach, as the simple crossover patently fails to preserve the permutation except in very fortunate circumstances. For instance, the offspring of the parents in the following case are clearly 'illegitimate':

```
P1   2 1 3 4 5 6 7      O1   2 1 3 2 7 1 5
           X
P2   4 3 6 2 7 1 5      O2   4 3 6 4 5 6 7
```

We need to ask here what it is that traditional crossover is doing that a sequence-based crossover should also do. Broadening the terms of Holland's schema analysis and the building-block hypothesis, we argue that it is to ensure that 'good' features of the parents can be passed on to the offspring, and to allow 'small' features of parents to join together to become 'large' ones.[2]

A number of different 'crossover' operators have been proposed for dealing with this situation, all of which try to maintain some feature of the parents. The PMX (partially mapped crossover) of Goldberg and Lingle [4], perhaps the most well-known, works as follows. Two crossover points are chosen, defining an interchange mapping. Thus in the example above, PMX might give

```
P1   2 1 3 4 5 6 7      O1   4 1 6 2 7 3 5
        X     Y
P2   4 3 6 2 7 1 5      O2   2 6 3 4 5 1 7
```

Here the crossover points X and Y define an interchange mapping $\{3 \leftrightarrow 6, 4 \leftrightarrow 2, 5 \leftrightarrow 7\}$.

[2] It is possible to extend the concept of a schema to problems of this type, as in Goldberg [14], but the meaningful features for permutation problems depend to a much greater extent on the specific problem than they do in a traditional binary GA.

authors (e.g. Prosser [12], Smith [17]), without apparently acquiring a definitive name. Oliver *et al.* [6] and Davis [36] also describe generalizations of the simple procedure described here. In its basic form, C1 chooses a crossover point X randomly, takes the 'left-hand' section of the first parent, and fills up the chromosome by taking in order each unassigned element from the second parent. For the example above it might generate the following offspring:

```
P1   2 1 3 4 5 6 7      O1   2 1 4 3 6 7 5
        X
P2   4 3 6 2 7 1 5      O2   4 3 2 1 5 6 7
```

The rationale for such operators as C1 and PMX is that they preserve the absolute positions in the sequence of elements of one parent, and/or the relative positions of those from the other. It is conjectured that this provides enough scope for the offspring to inherit useful features of the chromosome(s) without excessive disruption.

Perhaps because there is so much which seems problem-specific in permutation problems, the invention of recombination operators for such problems has proliferated. Fox and McMahon [18] discuss several such operators in detail, but many others have been proposed [19, 20, 21, 22]. In fact, this type of coding is perhaps the most well-researched of any in the area of combinatorial optimization, and operators such as PMX and C1 appear to be quite robust, and to operate reasonably effectively on a variety of permutation-coded COPs.

Permutation coding might at first seem irrelevant to the solution of bin-packing problems, but in fact Prosser [12] and Smith [17] have shown that such an approach can produce quite promising results, as we shall now discuss at greater length.

4 Hybridizing the GA

In the work reported here, we take this third route: the GA is hybridized with an *on-line heuristic* in a similar way to that used in [12, 17]. By the latter term, we mean a procedure that takes objects one at a time, in a given order, and fits each new object into a bin if possible, otherwise starting a new bin. Two well-known algorithms of this type are *first-fit* (FF) and *best-fit* (BF): FF simply fits the current object into the first possible bin (the bins are checked in some fixed order), while BF is a little more sophisticated, in that it tries to find the bin which will leave the least remaining space. Garey and Johnson [2] give the following performance guarantees

$$V \leq \frac{17}{10} V_{OPT} + 2$$

where V is the value of the objective function found by FF or BF, and V_{OPT} is the optimal objective function value. In other words, the number of bins used by FF or BF will not exceed the optimal number by very much more than 70%. In practice, of course, they often do far better than this.

These can be further improved if the on-line aspect is ignored, and the objects are first re-ordered in descending size order. The resulting heuristics FFD and BFD (D for *descending*) have guarantees

$$V \leq \frac{11}{9} V_{OPT} + 4.$$

Further, there is at least one permutation of the objects (and in practice, often many more than one) such that FF and BF will lead to an optimal solution.

Other such procedures could be devised—perhaps the simplest is *next fit* (NF), where a bin is filled until the next object fails to fit, when the current bin is put aside and a new bin started.

than one permutation of the objects for which NF would find an optimum. However, one would intuitively expect that there would be relatively far fewer such permutations for NF than for FF or BF.)

From a GA perspective, it is evident that we can re-interpret the bin-packing problem as a permutation problem, where we need to find a sequence such that the on-line heuristic gives an optimal solution. Such a re-interpretation means that we can immediately make use of the various GA procedures devised to deal with sequencing problems, which, as discussed above, have proved reliable and effective in a number of instances.

Falkenauer [26] has suggested that this approach might be expected to run into difficulties in finding good building-blocks. In essence, he argues that by interposing an on-line heuristic, the interpretation of what constitutes a good building-block becomes too remote from the actual coding used. In particular, he argues that the 'tail' part of a good string, which makes excellent sense in terms of its own 'head', may fail to do so in the context of the head of another.

This argument cannot be dismissed lightly. However, the interpretation of what is the 'head' and 'tail' after a sequence-based crossover is not entirely clear. Further, this sort of difficulty is not unique: in fact it can be argued that in almost every GA there is an intermediate 'algorithm' which is needed to decode the string into a solution in terms of the original problem specification. (In the standard GA terminology, the *genotype* needs to be decoded in order to obtain the *phenotype*.) In many traditional GAs, the 'algorithm' involves splitting a concatenated string into its component parts, then decoding the value of each part from its binary representation. The problem of remoteness of building-blocks can equally apply here, and related criticisms have been made elsewhere (see, for example Caruana and Schaffer [27] who argue for Gray-coded rather than binary-coded strings), yet binary-coded GAs have proved successful in a large variety of problems.

It is also true that the problem of redundancy in the coding still arises: it is possible for more than one coding to represent the same solution, and there is no guarantee that the recombination of two permutations leading to 'good' solutions will generate a permutation which also leads to a 'good' solution. For example, given 9 objects of size $\{5, 15, 25, 30, 35, 40, 45, 50, 55\}$ units and bins of size 100 units, the permutations $\{7, 4, 3, 8, 2, 5, 9, 6, 1\}$ and $\{9, 6, 1, 4, 3, 7, 2, 5, 8\}$ used with FF result in the same 3-bin solution $\{(3, 4, 7), (2, 5, 8), (1, 6, 9)\}$. However, applying C1 after the 4^{th} item leads to the permutations $\{7, 4, 3, 8, 9, 6, 1, 2, 5\}$ and $\{9, 6, 1, 4, 7, 3, 8, 2, 5\}$: using FF with the first of these leads to the 4-bin solution $\{(3, 4, 7), (1, 6, 8), (2, 9), (5)\}$[3].

In spite of these potential problems, existing work [12, 17] did not suggest that such problems had been observed in practice. Further, a search of the literature showed that a similar approach had been tried for clustering and partitioning problems [20, 28], with reasonable success. Thus, in order to investigate more completely whether, and in what circumstances, such a hybrid approach was worthwhile, a series of experimental investigations was carried out.

The experiments used a GA coded in Pascal and running on a Sequent S82 computer, originally written for the $n/m/P/C_{max}$ machine sequencing problem. (Incidentally, it is one of the attractive features of a GA that the only program alterations needed were those needed to specify the fitness function.) This used an incremental generation replacement policy, ranking-based selection, C1 crossover, a population of 30 strings one of which was the FFD solution, and an adaptive mutation rate—full details can be found in [9].

The fitness measure adopted was that used in [13], as was the program used to generate problem instances. With this method, firstly n objects were generated with random sizes such that they fitted exactly into m bins; some randomly chosen objects were then reduced in size so that m bins were still needed, but leaving a certain percentage *leeway* to allow for more packings which would also generate the optimal number of bins. For the initial experiments the *leeway*

[3] The author is grateful to the anonymous referee who went to the trouble of providing this example.

generated for each value of *leeway*; in each case there were 50 objects to pack. With the code provided by Falkenauer and Delchambre [13], the ratio m/n was found to vary within the range $(0.3, 0.45)$. Both in [13] and in Martello and Toth [29] it is suggested that instances of problems with m/n in this range are likely to be hard, as most bins can only contain a few items.

4.1 Choice of heuristic

The first question we wished to investigate was whether the power of the genetic search was affected by the choice of on-line algorithm. Thus the hybrid GA was used with each of the on-line algorithms—NF, FF and BF—and in each case a maximum of 3000 fitness evaluations was allowed. Table 1 reports the percentage of instances (out of 30) in which the optimal solution was found at each value of *leeway*. For comparison, the percentage of optima found by FFD is also shown.

Table 1: Relative performance of the on-line algorithms: 50-object instances

Algorithm	*leeway* value			
	2%	3%	5%	10%
FFD	0	3	3	23
GA+NF	0	0	0	0
GA+FF	17	27	47	90
GA+BF	17	43	67	100

The GA+NF hybrid was (as expected) a very poor method, and was actually outperformed by FFD. On the other hand, the hybrids with FF and BF both considerably out-perform FFD, finding optimal packings even for some of the very difficult instances where FFD almost always failed. GA+BF also appears superior to GA+FF, which would seem to indicate that it is worth hybridizing with as good a heuristic as possible. By way of comparison, Falkenauer and Delchambre [13] reported results which appear qualitatively to be very similar to GA+BF, although the actual problem instances they used were of course different.

The average CPU times for solving a single instance are reported in Table 2. The more sophisticated heuristics, as expected, require more computational effort for the harder problem instances. However, as the difficulty reduces, finding an optimal solution becomes easier for FF and BF, so in many cases the maximum 3000 fitness evaluations were not needed. Falkenauer and Delchambre [13] report CPU times of the same order of magnitude for problems of a similar size, although as the computer platform used is different, as well as other differences in the algorithmic details, we can draw no firm conclusions as to the relative efficiency of the approaches.

Table 2: Average CPU times (seconds): 50-object instances

Algorithm	*leeway* value			
	2%	3%	5%	10%
GA+NF	21.8	21.9	22.0	22.1
GA+FF	39.7	37.1	37.1	7.6
GA+BF	52.2	49.1	42.1	7.1

The question naturally arises as to whether the GA is actually exploring the solution space in an efficient manner, or whether it is simply performing a random walk. To pose the question in another way: is the GA really finding some underlying pattern in the order in which the bins are packed, or has it just 'got lucky'? If the latter were true, comparable results would be obtained simply by applying FF or BF to 3000 random permutations. This was carried out initially for the 30 instances of 10% *leeway* with the results shown in Table 3.

Table 3: Random search: 50-object instances with *leeway* 10%

	% solutions $\leq k$ bins from optimal				
k	0	1	2	3	4
FF	0.00	37.44	96.38	99.92	100.00
BF	0.00	58.33	98.19	99.97	100.00

It can be seen that even for this, the easiest set of problem instances, no random permutation ever gave rise to the optimum, while there is a substantial probability that a random permutation cannot even achieve 'optimal+1'. In contrast, none of the GA+FF and GA+BF solutions were ever further than 1 bin from optimal, even when the optimum itself was not found. These experiments were repeated for the harder problem instances (and for the larger problem instances reported more fully later) but as expected after the results on the easiest set, no optimum was ever found by random search in the space of permutations. (Detailed tables of results have been omitted for reasons of brevity; suffice it to say that the only discernible qualitative change from the above results was a slight tendency for more solutions to be found within a given number of bins from the optimum as the *leeway* value increases.)

The fact that on the average BF performed much better than FF for these random permutations, although their worst-case performance is the same, also appears to confirm the suggestion above that BF *is* a better heuristic, so that to hybridize the GA with BF is probably a better strategy than with FF.

5 Problem size reduction

The results reported above are encouraging: it would appear that GA+BF in particular is competitive with the rather more complicated approach of [13], although it is not clearly better. However, it is possible further to improve the performance of these hybrid heuristics by exploiting the *population* aspect of a GA to reduce the size of the search space explicitly—an approach which has apparently not been used before.

Problem size reduction is a technique which is commonly applied in classical approaches to solving COPs such as Lagrangean relaxation and branch-and-bound. Simply, the concept is to use information obtained from attempts to find upper or lower bounds to deduce that certain variables must take particular values in an optimal solution. Such variables can then be fixed in subsequent investigations, so that the size of the remaining problem is reduced. In the context of bin-packing, Martello and Toth [29] describe a general procedure which exploits an IP formulation. The heuristic approach described below is quite different from theirs, although related in spirit, and is used primarily because it is easy to implement in a GA context.

We assume that if we can find any subset I' of the objects such that

$$\sum_{i \in I'} c_i = C_{max}$$

whenever this event is detected, these objects are deleted and the heuristic attempts to solve the (smaller) problem that remains. In general, the above requirement could perhaps be relaxed to

$$\sum_{i \in I'} c_i \geq (1 - \epsilon) C_{max},$$

where ϵ is a small positive number.

A further benefit arises when this is carried out in the context of a GA: the fact that there are M solutions in the population means that on the detection of such an event in *any* member of the population, the identifiers of the objects in I' can be 'broadcast' to *all* the permutations in the current population. After deleting the objects concerned from every permutation, the GA can resume on a now smaller problem.

It should be noted that by employing this heuristic reduction rule, there is a risk of ensuring sub-optimality. For instance, given 6 objects of sizes $\{3, 3, 4, 6, 7, 7\}$, with a bin capacity of 10 units, the optimal packing uses 3 bins. Yet it is clear that if the first 3 items are packed into a bin, the remaining items cannot be packed into 2 bins. However, it was decided to ignore this potential problem, on the assumption that it would occur only very rarely in practice.

This approach was implemented with a value of $\epsilon = 0.00001$ for the 120 problem instances described in the previous section, for the GA hybrids with FF and BF. It was decided not to try reduction on GA+NF, given its poor performance in the initial experiments. However, a comparison was made with the branch-and-bound (B&B) scheme of Martello and Toth [29]. This is an exact method which in some cases finds optimal solutions fairly quickly. However, it is also possible for B&B to run for a very long time without reachng the optimum, and some form of stopping criterion is needed. In this case B&B was run for the same amount of computer time as either GA+FF+R or GA+BF+R, whichever was the longer for a given group of instances. The results were as shown in tables 4 and 5.

Table 4: Performance using reduction: 50-object instances

Algorithm	*leeway* value			
	2%	3%	5%	10%
B&B	37	27	50	90
GA+FF+R	93	90	90	97
GA+BF+R	80	90	97	97

There is a marked improvement in the performance of both hybrids when size reduction is incorporated (although it is interesting that in one of the easiest problems, neither hybrid found the optimum whereas previously GA+BF had done so—possibly for the reason suggested by the above example, that the remainder problem had too many 'large' objects left). It appears that there is really no longer any major difference between using FF and BF either, and both find more optimal solutions in the given amount of time than the exact approach. Again, although a direct comparison was not possible, they also appear to find the optimum more often than the procedure of Falkenauer and Delchambre.

The CPU times needed were also dramatically less than the standard heuristics; the reduction process inevitably incurs some overhead, but because much of the time is then spent in solving smaller problems the overall effect is highly beneficial.

Table 5: Average CPU times (seconds) using reduction: 50-object instances

Algorithm	*leeway* value			
	2%	3%	5%	10%
GA+FF+R	3.4	4.8	2.2	2.3
GA+BF+R	4.0	3.1	2.1	1.9

5.1 The influence of the crossover operator

All the above procedures were repeated using PMX instead of C1; as the results were hardly distinguishable, full details have been omitted here.

5.2 The effect of the number of bins m

Another factor that may appear relevant is the number of bins m: as indicated above, the values of m generated by the Falkenauer/Delchambre code varied between $0.3n$ and $0.45n$. If m increases much beyond this, many bins are likely to have only a single item, and 'packing' becomes somewhat of a misnomer! For smaller values of m there are far more small objects and packing becomes relatively easy. These limits thus probably define quite well most of the problem instances that are going to prove difficult. Within this range, however, it would be interesting to know if the level of difficulty varied with m.

To this end, the performance of each heuristic on each set of 30 problems was classified against the number of bins required in the optimal solution, and a simple χ^2 test carried out to test the hypothesis that the proportion of 'successes' (cases where the optimum was found) was independent of m. In some cases it was not possible to carry out a test, since in either all or none of the instances was the optimum found. However, in those cases which could be tested, the hypothesis was never rejected. It would seem therefore that at least for ratios of m/n in the range $(0.3, 0.45)$, the performance of the heuristic was not influenced by its actual value.

6 Some larger problems

Next, some experiments were carried out with 100-object problem instances (again, 30 instances for each value of *leeway*), in order to evaluate the performance of the heuristics further. All GA parameter settings were the same, and the results were as shown in Tables 6 and 7. B&B was again given the same time as the maximum of GA+FF+R and GA+BF+R. (Once more, owing to its poor performance on the smaller problem instances, the NF heuristic was not used.)

The general pattern of performance is similar to that in the 50-object cases, apart from a slight tendency for BF+R to perform better than FF+R both in terms of solution quality and computational effort.

However, these problem instances are clearly harder to solve, not just because there are twice as many objects, but because for a given value of *leeway*, there will also be on average twice as many bins in the 100-object instances (the 50-object instances needed ~ 20 bins, whereas the 100-object instances needed ~ 40). Thus, for a given *leeway* value the bins will have to be much more tightly packed in an optimal solution, and we can expect the number of optimal solutions to be far fewer. In the light of this, it is encouraging that the heuristics which incorporated reduction were still able to find optima with a reasonable probability, even for the cases of 2% leeway—which for these instances meant that the *average* bin was about 99.95% full in the optimal solution.

Table 6: Relative performance: 100-object instances

Algorithm	*leeway* value			
	2%	3%	5%	10%
FFD	0	0	0	10
B&B	0	3	7	67
GA+FF	0	3	7	70
GA+BF	0	0	3	87
GA+FF+R	23	30	53	77
GA+BF+R	30	33	57	83

Table 7: Average CPU times (seconds): 100-object instances

Algorithm	*leeway* value			
	2%	3%	5%	10%
GA+FF	144	143	133	59
GA+BF	193	192	183	53
GA+FF+R	29	29	26	20
GA+BF+R	22	22	20	12

Thus the experiments with 100 objects are not very informative about the relative difficulty caused simply by having more objects. Further sets of 30 problem instances were generated with 100, 200 and 500 objects, in which the amount of *leeway* was systematically varied from a value for which almost no optima were found to a value for which nearly all optima were found. In this way it was hoped that a general view could be formed as to the circumstances in which the GA-based hybrids could be expected to perform satisfactorily. In the case of 500 objects only the reduction-based heuristics were run, since the amount of computation for the standard hybrids was becoming excessive. Once more the amount of time allowed to the B&B was the greater of that needed by GA+FF+R or GA+BF+R.

The results are shown in Figures 1-4, where the percentage of optimal solutions found by each method is plotted against the *leeway*. (FFD was also run for a comparison, but as the curves were far below all the others, and in order not to clutter up the graphs unnecessarily, these results have been omitted.) The relevant variations in computer time required are also shown in Figures 5-8.

A general discussion of the implications of these results is deferred until section 7, so that we can first briefly mention some further analysis of the effect of m, and then describe some experiments with a different class of problem instances.

6.1 The effect of m

The effect of m was again explored for each set of problem instances by means of a χ^2 test of the hypothesis that the proportion of successes is independent of m. In two cases (GA+BF+R, $n = 100$, *leeway* per bin = 0.1% and 0.15%) the χ^2 value was marginally significant at the 5% significance level. In each case it appeared that the heuristic had relatively more successes on problem instances with larger values of m. However, on the whole it would be hard to conclude that for the range of values considered, m has a significant effect on the performance of the GA-based heuristics.

In their experimental work, Martello and Toth [29] evaluated several classes of problem instance using their exact algorithm. Class 1 objects had integer sizes drawn from a $U(1,100)$ probability distribution, class 2 were distributed $U(20,100)$ and class 3 $U(20,80)$. Within each class bin sizes were either 100, 120 or 150 units. They found the most consistently difficult problem instances were class 2, with a bin size of 150 units.

In a further evaluation of the GA hybrids, we followed [29] by generating 20 instances of such problems for the case of 200 and 500 objects. The class 2(150) problems did indeed prove difficult for the exact algorithm: even when 100,000 backtracks were allowed, most cases were not solved to optimality. However, nearly all the instances in Martello's and Toth's other classes were usually solved quickly by branch-and-bound—no longer than it took for the GA took to evaluate its initial population! It was thus clearly of little value to use a GA-based approach for problems other than in the class 2(150) case.

In order to compare their relative performance of the exact algorithm and the GA-reduction hybrids, the following experiments were carried out: first the reduction-based heuristics were applied using the same parameters as in the earlier experiments. Then the exact algorithm was run for the same amount of computer time as GA+BF+R. Finally the solutions were compared against the obvious lower bound

$$LB = \lceil \sum_{i=1}^{n} c_i / C_{max} \rceil$$

where object i has size c_i and the bin size is C_{max}. The results in Table 8 show the frequency with which the distance of the best solution of each method from LB was k bins.

Table 8: Comparison of heuristics with B&B: M&T class 2(150) instances

	% solutions $\leq k$ bins from LB			
k	0	1	2	3
200-object instances				
FFD	0	75	100	100
B&B	55	100	100	100
GA+FF+R	70	100	100	100
GA+BF+R	70	100	100	100
500-object instances				
FFD	0	0	45	100
B&B	0	35	85	100
GA+FF+R	5	85	100	100
GA+BF+R	10	85	100	100

Another way of comparing performance is in terms of the 'head-to-head score' of the methods, where the number of instances in which one method outperforms the other is counted. These figures are reported in Table 9.

7 Discussion

7.1 Falkenauer's problems

If the problem instances generated by the Falkenauer and Delchambre code are easily solved by B&B, it would suggest that however interesting from a GA perspective, the GA-based heuristics

Table 9: Head-to-head scores: M&T class 2(150) instances

n	200	500
FFR-B&B	5-2 (13=)	15-0 (5=)
BFR-B&B	6-2 (11=)	14-0 (6=)

are unimportant as a practical approach. It was thus essential to compare the effectiveness of the heuristics for these problem instances with the exact method. In case it might be that the B&B procedure simply needed more time, all runs were repeated with 10 times the time allowed for the original results (this was far longer than any of the heuristics were ever allowed). It was noteworthy that the proportion of optima found barely changed in any of these cases. It would appear that unless the exact method finds the optimum quickly, it is unlikely to do so in any reasonable amount of computing time. It was clear therefore that B&B does find these particular problem instances quite hard to solve, at least for small values of *leeway*, so that the search for an effective heuristic method is justified.

Overall, the results show that hybridizing a permutation-coded GA with an on-line heuristic produces excellent results in certain circumstances. From the graphs of Figures 1-4 we can see the variation of performance more clearly. It is evident that over a range of values of *leeway*, most of the GA-based heuristics have a higher probability of finding an optimal solution than B&B. However, no one heuristic is consistently the winner. For instance, using reduction gives better results on the smaller problem instances, but for the larger tighter ones it performs worse. A possible explanation for this might be that the GA-based methods needed more time to solve the larger instances, so the runs were repeated with 30000 fitness evaluations instead of 3000: GA+FF+R did improve to around the same level as GA+FF, but now took about twice as long, while GA+BF+R did not improve at all. While these methods were much faster than the standard hybrids, this tendency to converge to sub-optimal solutions in the relatively tight problems suggests that the hope expressed in Section 4 that induced sub-optimality would seldom arise seems not to be well-founded in general.

Again, while GA+BF was superior to GA+FF for the smaller problems, the situation appeared to reverse for the larger ones. The reason for this is not clear, although an explanation could be attempted on similar lines to that for the deterioration in performance of the reduction-based hybrids. Because the BF heuristic aggressively searches for a bin which it can fill to (or close to) its maximum, it also has the potential to induce sub-optimality at an early stage.

The graphs for average CPU times are rather more obvious: using BF typically takes somewhat longer than FF, since the BF search necessarily examines all the bins on each iteration. Reduction clearly speeds up the search dramatically for both FF and BF; however, the gradients of the curves are different, so that while GA+BF+R is quicker for small values of *leeway*, when it presumably finds more deletions, GA+FF+R is faster for larger values, when deletions are more easily found, and the relatively superior speed of FF has the advantage in solving the remainder problem.

A surprising feature of these results which is not observable from the graphs was that, in the more difficult cases, the discovery of an optimal solution by one method was fairly independent of its discovery by another. Thus, a good strategy would be to run each heuristic separately on a given instance and choose the best solution. However, when the constraint becomes looser (i.e. greater *leeway*) and the probability of finding an optimum approaches 100%, it must be conceded that the B&B approach is much more efficient than even the quickest of the GA-based heuristics, and should undoubtedly be preferred.

It seems doubtful whether this approach can solve to optimality large problem instances (i.e.

exact method of Martello and Toth found it even harder to achieve optimal solutions for tight instances with 200 and 500 objects, so we would conclude that even larger tight problems are likely to be very hard to solve by any method. (Whether Falkenauer's approach would do better is difficult to judge, as he did not report any experiments on problem instances as large as those considered here.)

7.2 The class 2(150) problems

Falkenauer's problems are useful in that it is easy to construct problem instances with known optimal solutions. However, it is a peculiar class in certain respects—for example, in the optimal solution all the bins except one are filled exactly. (This might be thought to favour the reduction-based approach, since well-filled bins are precisely what it is looking for.) Another feature of Falkenauer's problems was that in all the experiments carried out FFD never did *worse* than 'optimal+1', whereas for the M&T class 2(150) case, FFD often failed to do *better* than 'optimal+1'. Thus although it is usually hard to find the optimum of Falkenauer's tight problems, it appears to be quite easy to get to within one bin of it.

The class 2(150) problems of [29] have rather different characteristics. There are no really small objects, and many large ones, so that to be able to fill a bin exactly was fairly uncommon. The optimal solution (when it was found) was not always identical to the lower bound, so that the *leeway* could not always be determined. For instances with known optima, *leeway* was measured and found to vary from 1% to 98%. In the case of the 200-object instances the value of *leeway* did not seem to influence the ease or otherwise of finding the optimum. For example, there was no statistically significant difference (P=0.26) between average *leeway* values for the cases where B&B did or did not find the optimum. This is clearly different from the experience with Falkenauer's problems. (Too few optima were found for this question to be considered in the case of 500 objects.)

As remarked above, FFD not infrequently produced solutions which were at least 2 bins worse than optimal (or, in cases where the optimum was not found, at least 2 bins worse than the best known solution). Thus 'good' heuristic solutions to these instances are not easily generated, and optimal solutions cannot be found in a reasonable amount of computer time. In such circumstances, more effective heuristics are clearly needed.

It would appear from the results shown in Tables 8 and 9 that the GA-based hybrids were able to outperform the B&B approach given the same amount of computer time, for this particular class of problems. There is no significant difference between using FF and BF in terms of solution quality, although GA+BF+R took over a third more computer time than GA+FF+R.

8 Conclusions and extensions

This paper has reviewed some of the difficulties associated with the implementation of genetic algorithms for the solution of combinatorial optimization problems, and has suggested that hybridization is a possible route of improvement. It has also been argued that the population perspective of a GA should enable full advantage to be taken of problem size reduction techniques.

Experimental results have been presented which show that hybridizing the GA with an on-line heuristic is indeed a promising approach to the solution of one such problem—bin-packing—that is difficult for traditional pure GA approaches, albeit that the application of these hybrid methods would in practice be restricted only to problem instances which cannot be

[4] We were precluded from attempting problem instances of this magnitude by restrictions on computer space and time.

optimal solutions more reliably than an exact method over a range of values of *leeway*. The results tended to suggest what one would intuitively expect, that the choice of on-line heuristic is important—using a poor heuristic like NF gives the GA little chance. However, what is the 'best' choice seems to depend on certain problem-specific parameters; in particular further investigation is needed in an attempt to ascertain why BF seemed to give worse results for the larger problems.

When a problem size reduction heuristic was incorporated, the performance improved further for small problems, both in terms of the probability of finding an optimum, and in terms of the speed with which it can be accomplished. However, for larger tight problems, although the speed advantage remained, the solution quality deteriorated. It was pointed out that the heuristic reduction technique used here could lead to sub-optimality by prematurely eliminating certain objects, although we assumed that this would occur rarely. The performance of the heuristic (relative to the procedures that did not use reduction) was such that this assumption could not be sustained in general, although partially justified for smaller problems.

A possible way of alleviating this situation would be as follows: objects identified by the heuristic reduction procedure need not be completely eliminated, but could simply be tagged as members of a 'full' bin, which (at least temporarily) are to be ignored by the FF or BF heuristic. If an optimal solution is not obtained after a certain time they could be re-introduced into the object pool. This is similar in spirit to some of the ideas of recency-based tabu search, and raises similar questions as to how many cycles should pass before tagged objects are 'untagged' again. We have recently investigated a similar idea in the context of *graph partitioning*, where vertices of a graph have to be assigned to a subset in order to meet various criteria at minimum cost. Given two solutions to be genetically recombined, we separate the vertices into two subsets, one in which the vertices have fixed assignments obtained by reference to 'genetic' criteria, the other in which they are placed by an on-line heuristic. Results we have obtained using these hybrid methods [30] appear to be a significant improvement over existing GA approaches, and this approach is currently being applied to the case of bin-packing.

Another possibility for improving performance would be to combine the GA with a neighbourhood search procedure. This approach (becoming known as a 'memetic' algorithm [31]) has proved successful in a number of cases but has the drawback of excessive computational requirements. However, for those problems where branch-and-bound nearly always fails, this approach might be worth exploring.

It is noteworthy that several other combinatorial problems could be attacked in this way: the example of graph partitioning has already been mentioned. Another problem is *parallel machine scheduling*, where the objective is to minimize the *makespan* (the time at which the last job is completed). A simple on-line heuristic for this problem is to schedule each job on the machine that will produce the earliest completion time. It is also easy to find a lower bound on the makespan, and a plausible heuristic reduction rule is thus to remove jobs allocated to a machine for which the completion time is 'close' to the lower bound. Preliminary results on hybridizing these heuristics with a permutation-coded GA are promising, although sub-optimality again may be induced by the reduction procedure. Other problems which could be amenable to a similar approach include *assembly-line balancing*, *knapsack* and *generalized assignment*.

Further, given that a heuristic problem size reduction has performed fairly well in the case examined here, it would be interesting to investigate the possibility of combining a genetic algorithm with a more classical approach to optimization which uses a truly rigorous approach to reduction. For instance, it is possible to interpret a GA as a type of implicit tree-search, but unlike branch-and-bound techniques which explicitly partition the search space into disjoint regions, a GA uses a collection of disparate solutions to try to direct the search into regions inhabited by 'good' solutions. While initially the search regions are randomly dispersed, as

character, as certain schemata are accepted as defining good search areas. If the building-block hypothesis holds, then these search areas become more and more tightly defined, rather as in a traditional hierarchical tree-search procedure. However, the information used by the GA in order to intensify its search is not 'hard' information which can fix some variables at their optimal values before an implicit enumeration is complete, but 'soft' estimates of average fitness in a particular region. Thus, integrating hard information into a GA could possibly help the algorithm concentrate its search more effectively in 'promising' regions of the search space.

We should also mention a subordinate line of enquiry suggested by this research: the question of how one defines and generates 'good' benchmark problem instances for bin-packing. The concept of *leeway* is clearly one parameter of importance, but the experience of the Martello and Toth class 2(150) cases suggests that it is not the only one. The value of the ratio m/n must also have an influence, as must the mean size (and probably also the variance) of the objects.

In conclusion, this paper has shown that hybridization of GAs is potentially a very fruitful approach to the solution of hard combinatorial optimization problems, and has suggested a number of avenues for further exploration of such procedures.

Acknowledgement

The author would like to thank Emmanuel Falkenauer for making available his C program for generating instances of bin-packing problems.

References

[1] R.M.Karp (1972) Reducibility among combinatorial problems. *In* R.E.Miller and J.W.Thatcher (Eds.) *Complexity of Computer Computations*, Plenum Press, New York.

[2] M.R.Garey and D.S.Johnson (1979) *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W.H.Freeman, San Francisco.

[3] C.R.Reeves (Ed.) (1993) *Modern Heuristic Techniques for Combinatorial Problems*. Blackwell Scientific Publications, Oxford.

[4] D.E.Goldberg and R.Lingle (1985) Alleles, loci and the travelling salesman problem. *In* [32].

[5] J.J.Grefenstette (1987) Incorporating problem-specific knowledge into genetic algorithms. *In* L.Davis (Ed.) (1987) *Genetic Algorithms and Simulated Annealing*. Morgan Kauffmann, Los Altos, CA.

[6] I.M.Oliver, D.J.Smith and J.R.C.Holland (1987) A study of permutation crossover operators on the traveling salesman problem. *In* J.J.Grefenstette(Ed.) (1987) *Proceedings of the* 2^{nd} *International Conference on Genetic Algorithms*. Lawrence Erlbaum Associates, Hillsdale, NJ.

[7] D.Whitley, T.Starkweather and D.Shaner (1991) The traveling salesman and sequence scheduling: quality solutions using genetic edge recombination. *In* [36].

[8] G.A.Cleveland and S.F.Smith (1989) Using genetic algorithms to schedule flow shop releases. *In* [33].

[9] C.R.Reeves (1993) A genetic algorithm for flowshop sequencing. *Computers & Ops.Res.*, (to appear).

genetic algorithms with penalty functions. *In* [33].

[11] A.Kapsalis, V.J.Rayward-Smith and G.D.Smith (1993) Solving the graphical Steiner tree problem using genetic algorithms. *J.Opl.Res.Soc.*, **41**, 397-406.

[12] P.Prosser (1988) A hybrid genetic algorithm for pallet loading. In *Proceedings of 8th European Conference on Artificial Intelligence*. Pitman, London.

[13] E.Falkenauer and A.Delchambre (1992) A genetic algorithm for bin packing and line balancing. In *Proceedings of the IEEE International Conference on Robotics and Automation*.

[14] D.E.Goldberg (1989) *Genetic Algorithms in Search, Optimization, and Machine Learning.* Addison-Wesley, Reading, Mass.

[15] J.H.Holland (1975) *Adaptation in Natural and Artificial Systems.* University of Michigan Press, Ann Arbor.

[16] C.R.Reeves (1994) Genetic algorithms and combinatorial optimization. In V.J.Rayward-Smith (Ed.) *Applications of Modern Heuristic Techniques*. Alfred Waller Ltd, Henley-on-Thames, UK.

[17] D.Smith (1985) Bin packing with adaptive search. *In* [32].

[18] B.R.Fox and M.B.McMahon (1991) Genetic operators for sequencing problems. *In* G.J.E.Rawlins (Ed.) (1991) *Foundations of Genetic Algorithms.* Morgan Kaufmann, San Mateo, CA.

[19] T.Starkweather, S.McDaniel, K.Mathias, D.Whitley and C.Whitley (1991) A comparison of genetic sequencing operators. *In* [34].

[20] J.N.Bhuyan, V.V.Raghavan and V.K.Elayavalli (1991) Genetic algorithm for clustering with an ordered representation. *In* [34].

[21] J.L.Blanton Jr. and R.L.Wainwright (1993) Multiple vehicle routing with time and capacity constraints using genetic algorithms. *In* [35].

[22] A.Homaifar, S.Guan and G.E.Liepins (1993) A new approach on the travelling salesman problem by genetic algorithms. *In* [35].

[23] N.J.Radcliffe (1992) Non-linear genetic representations. *In* R.Männer and B.Manderick (Eds.) (1992) *Parallel problem-Solving from Nature 2.* Elsevier Science Publishers, Amsterdam.

[24] J.Antonisse (1989) A new interpretation of schema notation that overturns the binary encoding constraint. *In* [33], 86-91.

[25] M.D.Vose and G.E.Liepins (1991) Schema disruption. *In* [34].

[26] E.Falkenauer (1992) *Personal communication.*

[27] R.A.Caruana and J.D.Schaffer (1988) Representation and hidden bias: Gray vs. binary coding for genetic algorithms. In *Proceedings of the 5th International Conference on Machine Learning*. Morgan Kaufmann, Los Altos, CA.

[28] D.R.Jones and M.A.Beltramo (1991) Solving partitioning problems with genetic algorithms. *In* [34].

tions. Wiley, New York.

[30] C.Höhn and C.R.Reeves (1994) Using genetic algorithms to solve graph partitioning problems. *J.Computer Aided Engineering*, (in review).

[31] N.J.Radcliffe and P.Surrey (1994) Formal memetic algorithms. *In* T.C.Fogarty (Ed.) (1994) *Evolutionary Computing: AISB Workshop, Leeds, UK, April 1994; Selected Papers.* Springer-Verlag, Berlin.

[32] J.J.Grefenstette(Ed.) (1985) *Proceedings of an International Conference on Genetic Algorithms and their applications.* Lawrence Erlbaum Associates, Hillsdale, NJ.

[33] J.D.Schaffer (Ed.) (1989) *Proceedings of* 3^{rd} *International Conference on Genetic Algorithms.* Morgan Kaufmann, Los Altos, CA.

[34] R.K.Belew and L.B.Booker (Eds.) (1991) *Proceedings of* 4^{th} *International Conference on Genetic Algorithms.* Morgan Kaufmann, San Mateo, CA.

[35] S.Forrest (Ed.) (1993) *Proceedings of* 5^{th} *International Conference on Genetic Algorithms.* Morgan Kaufmann, San Mateo, CA.

[36] L.Davis (Ed.) (1991) *Handbook of Genetic Algorithms.* Van Nostrand Reinhold, New York.

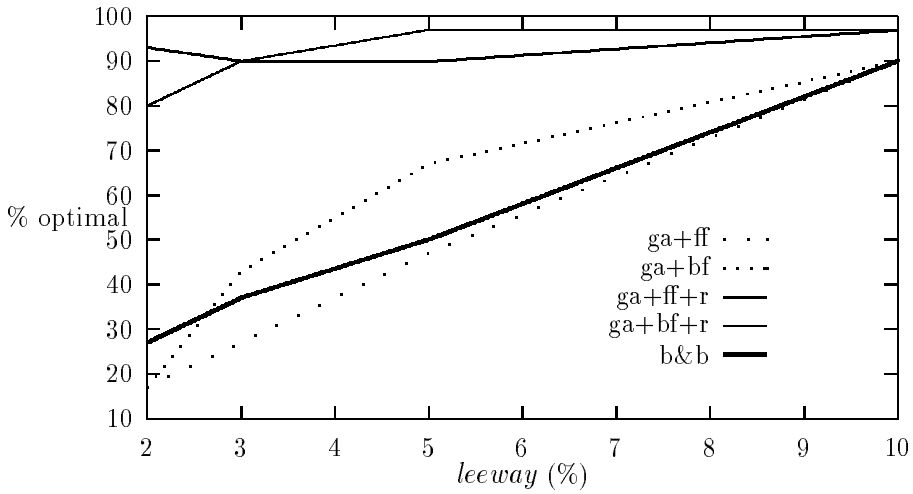

Figure 1: Performance of heuristics at different values of *leeway* : 50-object instances

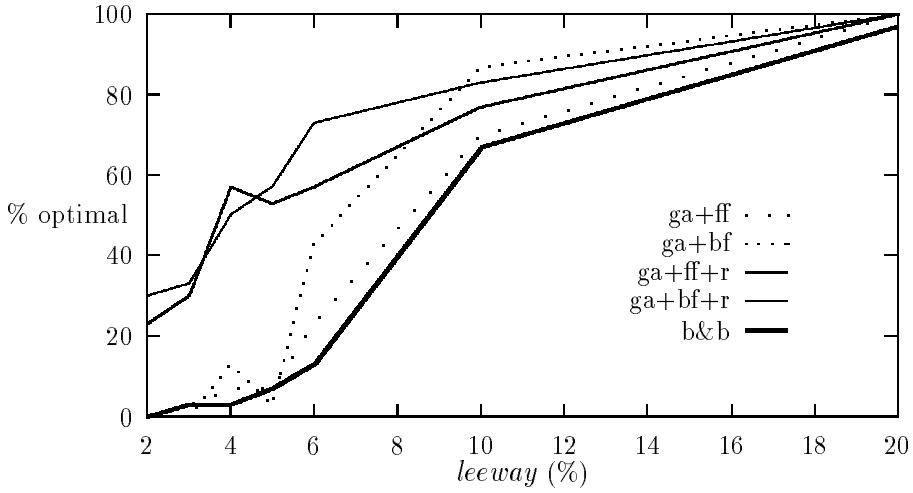

Figure 2: Performance of heuristics at different values of *leeway* : 100-object instances

Table 10: Relative performance: second set of 100-object instances

Algorithm	*leeway* value per bin			
	0.1%	0.15%	0.25%	0.5%
FFD	0	0	10	13
B&B	3	13	67	97
GA+FF	7	23	70	100
GA+BF	13	43	87	100
GA+FF+R	57	57	77	100
GA+BF+R	50	73	83	100

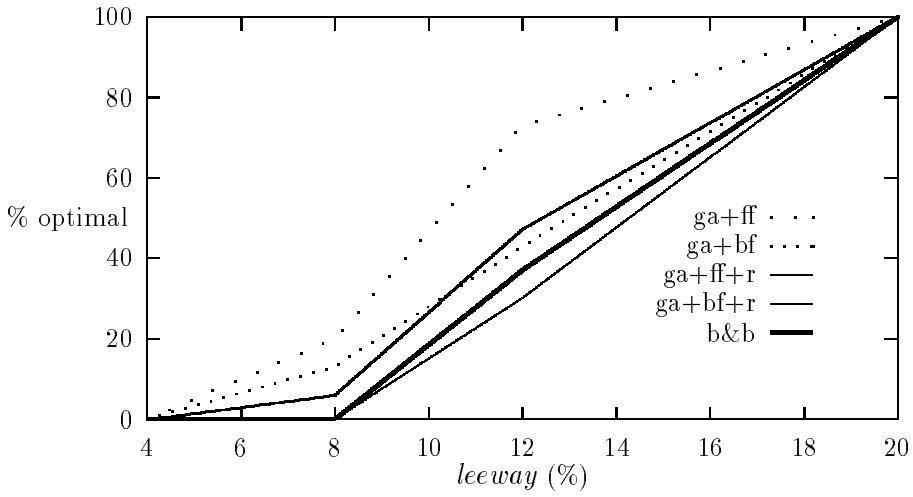

Figure 3: Performance of heuristics at different values of *leeway* : 200-object instances

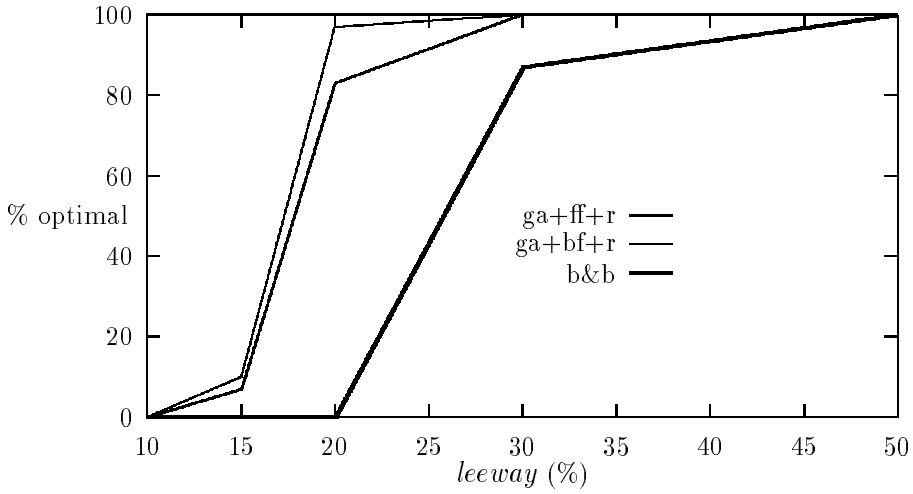

Figure 4: Performance of heuristics at different values of *leeway* : 500-object instances

Table 11: Average CPU times (seconds): second set of 100-object instances

Algorithm	*leeway* value per bin			
	0.1%	0.15%	0.25%	0.5%
GA+FF	118	104	58	3.4
GA+BF	152	121	53	3.7
GA+FF+R	22	23	20	4.0
GA+BF+R	17	15	12	4.8

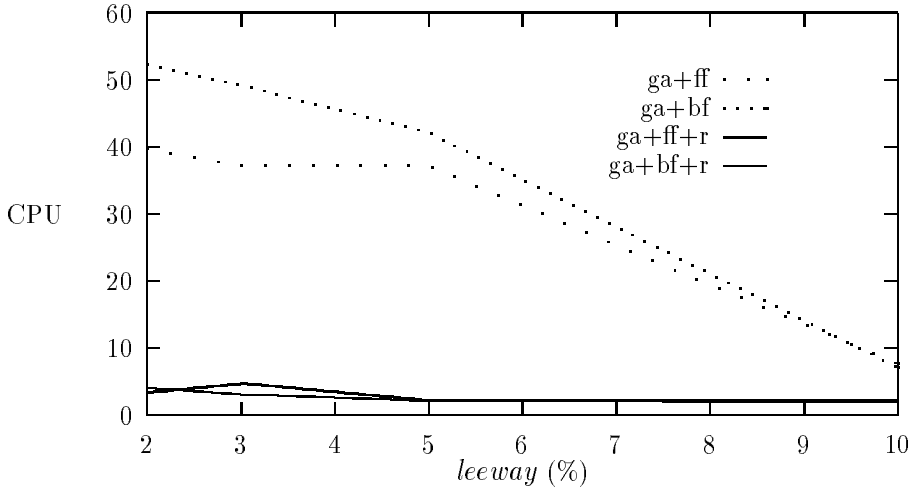

Figure 5: Average CPU time (sec) at different values of *leeway* : 50-object instances

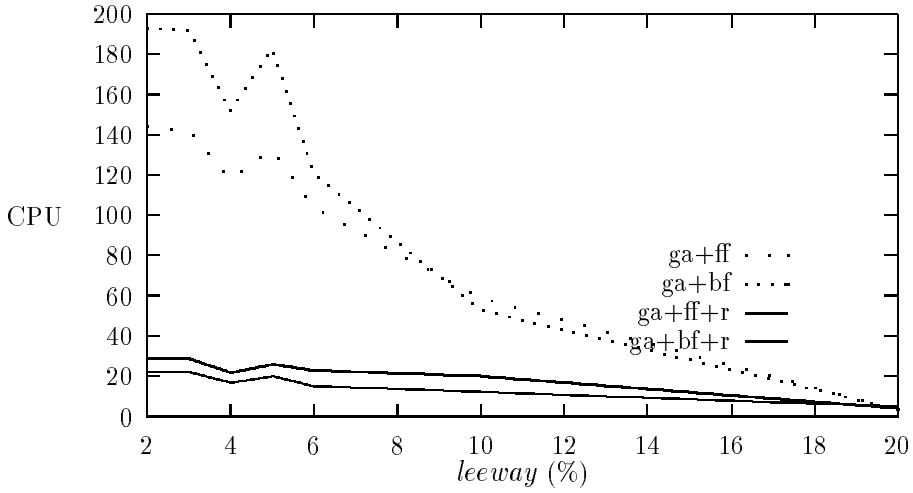

Figure 6: Average CPU time (sec) at different values of *leeway* : 100-object instances

Table 12: Relative performance: 200-object instances

Algorithm	*leeway* value per bin		
	0.1%	0.15%	0.25%
FFD	0	3	80
B&B	0	37	100
GA+FF	20	73	100
GA+BF	13	43	100
GA+FF+R	6	47	100
GA+BF+R	0	30	100

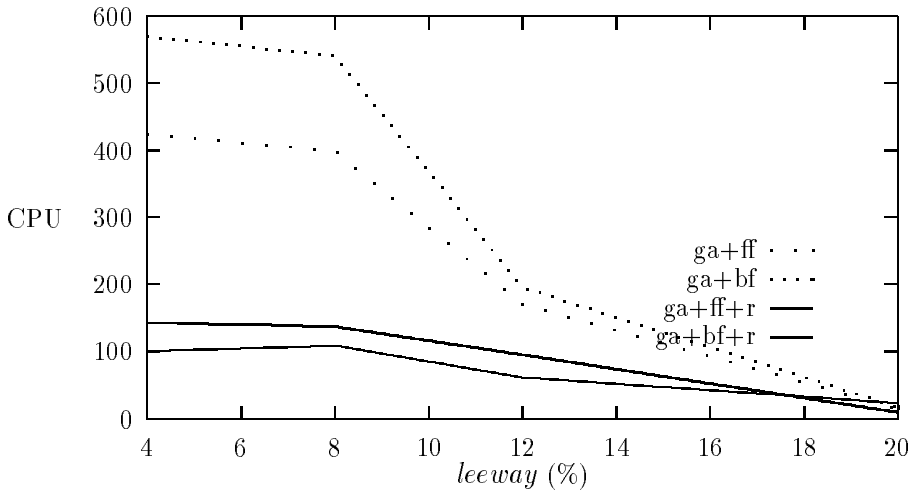

Figure 7: Average CPU time (sec) at different values of *leeway* : 200-object instances

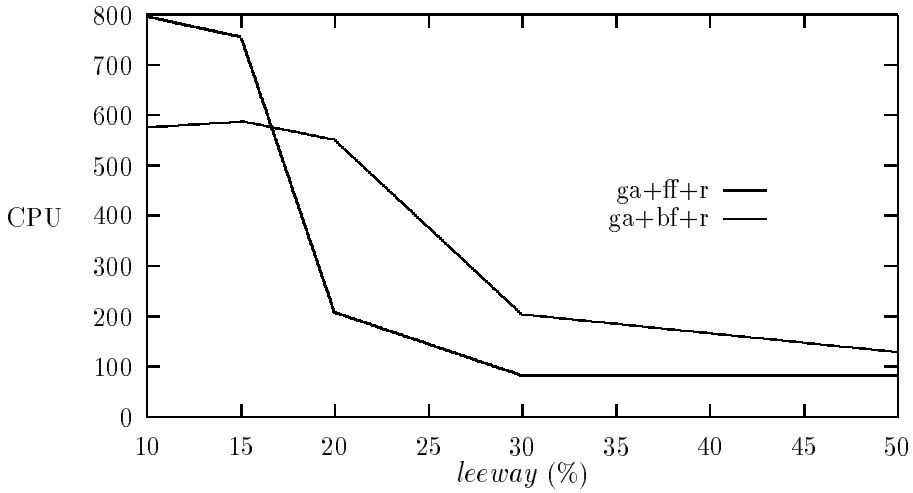

Figure 8: Average CPU time (sec) at different values of *leeway* : 500-object instances

Table 13: Average CPU times (seconds): 200-object instances

Algorithm	*leeway* value per bin		
	0.1%	0.15%	0.25%
GA+FF	399	171	14
GA+BF	541	196	17
GA+FF+R	137	95	10
GA+BF+R	109	61	23

Table 14: Relative performance: 500-object instances

Algorithm	*leeway* value per bin		
	0.05%	0.1%	0.15%
FFD	0	33	100
B&B	0	87	100
GA+FF+R	0	83	100
GA+BF+R	0	97	100

Table 15: Average CPU times (seconds): 500-object instances

Algorithm	*leeway* value per bin		
	0.05%	0.1%	0.15%
GA+FF+R	796	209	83
GA+BF+R	576	551	203